Procedia Computer Science
Volume 80, 2016, Pages 63–73
ICCS 2016. The International Conference on Computational Science

EMINENT: EMbarrassINgly parallEl mutatioN Testing

Pablo C. Cañizares[1], Mercedes G. Merayo[1], and Alberto Núñez[1]

Universidad Complutense de Madrid, Madrid, Spain.
pablocc@ucm.es, mgmerayo@fdi.ucm.es, alberto.nunez@pdi.ucm.es

Abstract

During the last decade, the fast evolution in communication networks has facilitated the development of complex applications that manage vast amounts of data, like Big Data applications. Unfortunately, the high complexity of these applications hampers the testing process. Moreover, generating adequate test suites to properly check these applications is a challenging task due to the elevated number of potential test cases. *Mutation testing* is a valuable technique to measure the quality of the selected test suite that can be used to overcome this difficulty. However, one of the main drawbacks of mutation testing lies on the high computational cost associated to this process.

In this paper we propose a dynamic distributed algorithm focused on HPC systems, called `EMINENT`, which has been designed to face the performance problems in mutation testing techniques. `EMINENT` alleviates the computational cost associated with this technique since it exploits parallelism in cluster systems to reduce the final execution time. In addition, several experiments have been carried out on three applications in order to analyse the scalability and performance of `EMINENT`. The results show that `EMINENT` provides an increase in the speed-up in most scenarios.

Keywords: Mutation testing, Scientific Computing, Parallel and Distributed Computing

1 Introduction

During the last years, the emergence of new technological trends, such as next-generation networks, always-connected mobile broadband and media services, has facilitated the rising of a new generation of IT. This emergence is characterised by high-speed and high-connectivity connection networks simultaneously used by millions of users. Similarly, there has been a rise of a new generation of applications and services, such as social networks and instant messaging applications, that allows users to share and process images, send posts and communicate with other users in a fast and accessible way. Thus, the growing popularity of these services, has lead to a massive data generation. For instance, in 60 seconds, WhatsApp users share 490,320 messages, Instagram users filter 216,000 images, Twitter users send 347,222 tweets and Facebook users share 2,246,000 posts [24]. In order to handle and process this massive amount of data, current systems have to face the challenge of performing these techniques efficiently and effectively.

doi:10.1016/j.procs.2016.05.298

Hence, it is important that communication networks achieve a high bandwidth with a low latency to transfer large amounts of data, while computational resources are exploited in parallel. One of the most used solutions to reduce long execution time is *High Performance Computing* (in short, HPC), a computational solution which provides an excellent price-performance ratio. The importance of this paradigm is reflected in the TOP-500 list, which shows that the 500 most powerful computers in the world are clusters [14].

Another important aspect that must be taken into account is the complexity of these services that are composed of diverse processes, such as text compression and image filtering. This fact hampers the validation process. Nevertheless, it is necessary to build adequate test suites to check their correctness before deploying them in the production environment. Currently, *testing* is the most often used technique to check the validity of software. One of the main difficulties in applying testing methodologies is to design an *appropriate* test suite. Mutation testing allows to improve the design of high quality test suites. This technique is based on applying *mutation operators* to programs that make small syntactic changes in order to produce a set of *mutants*. The idea is that if a test suite is able to distinguish between a program and the generated mutants, it should be good at detecting a faulty implementation. It helps to determine the effectiveness of a test suite and helps during the test generation. The effectiveness of a test suite is established on the basis of how many of the mutants it distinguishes from the original program. However, mutation testing is computationally expensive, since the number of mutants that are generated is huge and they must be executed against the test suite. In consequence, high computational power is required to speed-up the mutation testing process.

In this paper we propose `EMINENT`, a scalable, dynamic, and HPC-oriented algorithm [17, 15, 2], based on embarrassingly parallel computation ideas [9, 4]. `EMINENT` focuses on reducing the execution time associated to the classical mutation testing scheme. The proposed algorithm is *scalable*, the overall system performance increases as the computational resources. It is also *dynamic*, since the processed workload of each computational resource is assigned depending on its underlying characteristics. Moreover, `EMINENT` is focused on HPC and uses the shared resources of cluster systems to solve the same computational problem. This approach has been implemented using MPI, a standard Message-Passing Interface library to improve the communications in high performance environments, which properly fits with dynamic distribution strategies [5, 25].

The rest of the paper is structured as follows. Section 2 presents the state of the art of parallel mutation testing. Next, in Section 3 we describe `EMINENT` in detail. Section 4 presents some performance experiments by using three different applications. Finally, in Section 5, we present the conclusions and some lines of future work.

2 State of the art

Currently, several cost reduction techniques to improve the execution time of mutation testing can be found in the literature. These techniques are traditionally divided into three approaches: *do fewer*, *do faster* and *do smarter*. The proposal presented in this paper focuses on parallel testing, which is classified in the *do faster* approach. Although we have found some works in this research field during the last decades, it is worth mentioning that most of the proposals were introduced during the early nineties and the last 4 years.

The first contribution in parallel mutation testing can be traced back to 1988 with Mathur and Krauser [13]. In their approach, they proposed a novel technique to reduce execution costs using a vector processor. In this work, multiple mutants are simultaneously executed in a single processor using a sequence of vector instructions. Even though the approach greatly increases the computational performance of the mutation testing scheme, it is limited to mutants generated with scalar variable replacement operator. Afterwards, authors extended their work with a high performance approach based on shared-memory, called mutation unification, to support several existing mutation operators [11, 21]. In their studies, compilation was identified as a major bottleneck of the scheme. However, this issue can be alleviated by using current techniques that can be found in the literature [26, 12].

There exist multiple mutation testing frameworks that include parallel techniques to improve the performance and, consequently, to reduce the computational cost [10, 23]. Despite the benefits obtained by the use of Single Instruction Multiple Data improvements, these systems are limited by the number of processors that are comprised. Hence, it is necessary to include new distributed schemes of mutation testing that address this scalability issue.

In order to alleviate this problem, Offut et al. proposed the first mutation testing approach based on Multiple Instruction Multiple Data systems [16]. This work presents a parallel interpreter, called *HyperMothra*, which was implemented on a sixteen processor Intel iPSC/2 hypercube. In addition, diverse static schemes of distribution algorithms are included, such as distributing mutants in original order and distributing mutants randomly and uniformly by mutation type. The authors stated that the performance achieves almost a linear speedup over Mothra's sequential interpreter but they also identified the communications as the bottleneck of the system.

In the same line, Byoungju and Mathur presented the $P^{M}othra$ system [3]. This approach has a flexible architecture designed to provide a high degree of scalability. The system also provides the tester with a transparent interface to a distributed machine and includes a dynamic distribution algorithm that serves mutants to the available nodes. As in the previous proposals, the communication network is a bottleneck and slows-down the performance of the system.

Most recently, Mateo and Usaola have presented a study for adapting the existing cost reduction techniques to current technologies [20]. They introduce $Bacterio^{P}$, a parallel extension of the mutation testing tool *Bacterio* [19], using *Java-RMI* [6] in order to communicate the nodes of the network. In addition, the authors include five distribution schemes using dynamic and static distributions. Among these schemes, it is worth noting the *Parallel Execution with Dynamic Ranking and Ordering (PEDRO)* algorithm. This contribution is a dynamic distribution algorithm based on *Factoring Self-Schedulling* ideas [8], that considers to address the well known communication efficiency problem. Although this proposal achieves better results than the previous works, the mechanisms used in the communications are not the most adequate for high performance environments due to the high latency introduced by this technology. It has been shown that Java-RMI is 3 to 5 times slower than MPI [18]. Hence, we consider that the distribution process can be improved in order to achieve a higher level of parallelism by increasing resources efficiency.

In 2014, Saleh and Nagi presented the *HadoopMutator* framework. It is based on Map-Reduce programming model and distributes and executes mutant generation and testing process [22]. The framework is based on *Hadoop* engine and *Pitest* mutation testing framework. This approach follows a static schema in which the inclusion of dynamic distribution algorithms is not considered. Thus, this framework is not oriented to heterogeneous and dynamic environments.

3 EMINENT

Nowadays, there exist several techniques to improve the performance of the mutation testing process. In this paper we propose `EMINENT`, an algorithm to distribute the workload of this

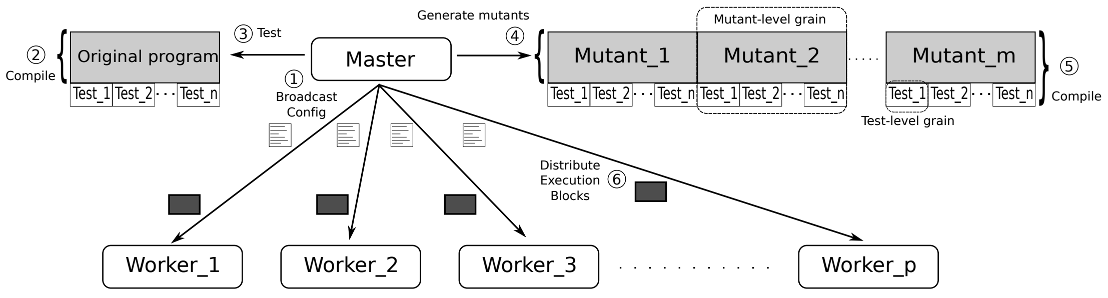


Figure 1: General scheme of `EMINENT`

process in order to reduce the execution time. The main goal of this work is to achieve an *scalable*, *dynamic* and *high performance* solution to face the computational challenges associated with mutation testing. Next, we describe the main features of `EMINENT`.

- *Scalable*: `EMINENT` has been designed to be deployed and executed in a distributed system. The increment in the quantity and quality of system's resources means the increase in its computational performance. `EMINENT` presents two types of scales: *horizontal and vertical*. The former allows to include more computing nodes to the system, while the latter allows to extend the computational resources in each node.

- *Dynamic*: In order to maximise the exploitation of computational resources, `EMINENT` splits the input dataset into blocks and dynamically delivers them to the available CPUs. Once a node finishes the execution of a block, it is provided with a new block until all the blocks have been processed. This distribution scheme benefits heterogeneous systems.

- *High performance*: The proposed algorithm is based on a *high performance* schema in which the shared resources of several machines are used as a whole to perform the mutation testing process. The testing process is executed in parallel over all the nodes of a cluster, taking advantage of the low latency communication network to maximise the parallelism and enhance the overall performance.

Algorithm 1 presents the main steps of `EMINENT`. The proposed scheme uses different processes. On the one hand the *master* process, that is responsible for orchestrating the algorithm. It splits the workload of the testing process in *execution blocks* and distributes them among the worker processes. On the other hand, the worker processes execute them and send the results back to the master process. The number of workers processes that are instantiated in the algorithm is variable and can be defined by the user.

Figure 1 shows the basic scheme of `EMINENT`. The first step consists in the selection of both the *source code* of the program to which the mutation testing process will be applied and the *test suite* that will be used during this process. Then, the master process compiles the original program ② and, if the compilation finishes successfully, the *testing process* begins. At this point, the master executes all the test cases in the selected test suite and stores the results ③. If the execution of all test cases is correct, the master invokes an external mutation testing tool to generate ④ and compile ⑤ all the program mutants. Mutants are produced by using *mutation operators* that aim to simulate common faults. Each mutation operator makes a small syntactic change in the source code. The execution of the generated mutants is distributed by

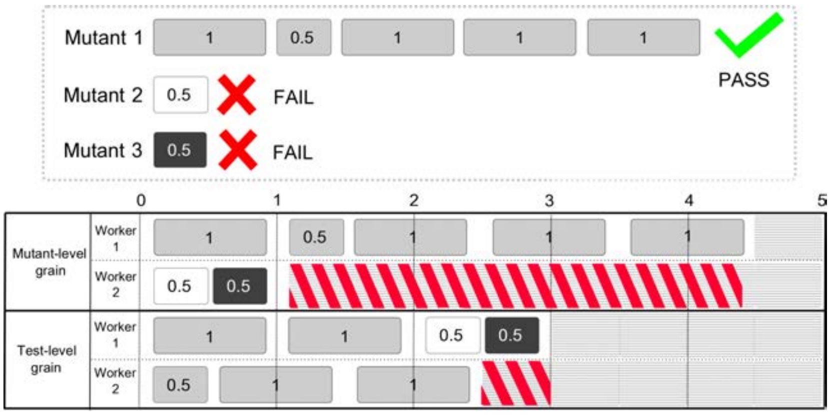


Figure 2: Comparison of different grain size (mutant vs test)

the master process among the the workers ⑥. For each mutant, the master process dispatches the test cases to the worker processes that will execute them against the mutant. The obtained results are sent to the master, that compares them with the ones produced by the original program. In the case that a difference is detected, the mutant will be considered *killed*, and all the running executions associated with it will be aborted and no more tests will be executed against it. The process continues until all the test cases are executed against all the mutants. Finally, the master process calculates the *mutation score* of the process, that indicates the percentage of killed mutants over the total number of mutants.

EMINENT uses a *test-level grain*, where the *execution blocks* are composed by a single test case, in contrast to the *mutant-level grain*, where the *execution blocks* are composed by the complete test suite, that usually is implemented in the dynamic distribution approaches existing in the literature. Figure 2 illustrates the difference between both of them when applied in the execution of mutant against a set of test.

Each algorithm is executed using 2 worker processes to test 3 different mutants against a test suite that contains 5 tests cases. The mutant 1 passes all the test cases in 4.5 units of time ($t_{c1} = 1, t_{c2} = 0.5, ..., t_{c5} = 1$) and mutants 2 and 3 fail the first test in 0.5 units of time. On the one hand, in the mutant-level grain algorithm the first worker executes all the tests over the mutant 1. Then, in parallel, the second worker executes, consecutively, the test 1 over the mutant 2 and 3. The total time elapsed during this process is 4.5 units of time. On the other hand, in the test-level grain scheme, the test cases can be executed by any of the workers. First, mutant 1 is executed against all the test cases. Then, when no more test cases have to be applied to mutant 1, worker 1 executes test 1 on mutants 2 and 3. In this case, the total time elapsed is 3 units of time. It is important to emphasize that the worker 2 is idle from $t = 1$ to $t = 4.5$ when using the mutant-grain level. However, this worker is only idle from $t = 2.5$ to $t = 3$ when the test-grain level is used. This difference means that the test-grain level is more adaptable to heterogeneous environments and allows to maximise the resources usage. As a consequence, the overall time of the testing process is reduced.

Algorithm 1 Parallel Mutation Testing

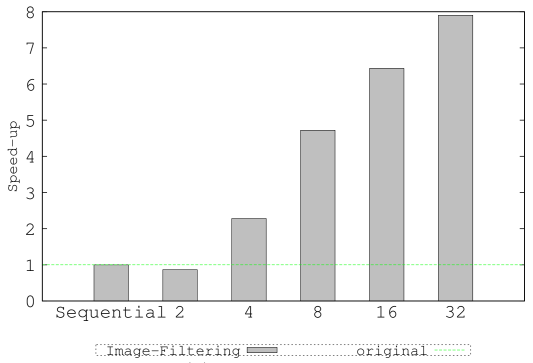
```
Require: config
1: MPI_Init();
2: numprocs ← MPI_Comm_size();
3: myId ← MPI_Comm_rank();
   // Master process
4: if (myId == MASTER) then
5:    originalResults ← executeTests(originalProgram, config.getTests());
6:    if areResultsCorrect(originalResults) then
7:       mutantList ← generateMutants();
8:       compileMutantsAndTests();
9:       MPI_Bcast (Config, MASTER);
10:      resultsMutants ← builtResultsTable (mutantList, config.getTests()); // Initial distri-
         bution among worker processes
11:      while (i<numProcs and getRemainingMutants (resultsMutants)>0) do
12:         currentMutant ← getCurrentMutant(resultsMutants);
13:         currentTest ← getCurrentTest (currentMutant, resultsMutants);
14:         execBlock ← buildExecBlock (currentMutant, currentTest, config.getExecBlock());
15:         MPI_Send (execBlock, i);
16:      end while// While there are remaining mutants ...
17:      while (continueProcessing) do
18:         result ← MPI_Recv (ANY, status);
19:         continueProcessing ← updateResults (result, resultsMutants);
20:         if (getRemainingMutants (resultsMutants)>0) then
21:            currentMutant ← getCurrentMutant(resultsMutants);
22:            currentTest ← getCurrentTest (currentMutant, resultsMutants);
23:            execBlock ← buildExecBlock (currentMutant, currentTest, con-
               fig.getExecBlock());
24:            MPI_Send (execBlock, status.MPI_SENDER);
25:         end if
26:      end while
27:   end if
      // Worker process
28: else
29:    while (continueProcessing) do
30:       MPI_Recv (execBlock, MASTER);
31:       result ← execute (execBlock);
32:       MPI_send (MASTER, result);
33:       continueProcessing ← execBlock.continueProcessing;
34:    end while
35: end if
```

4 Experiments

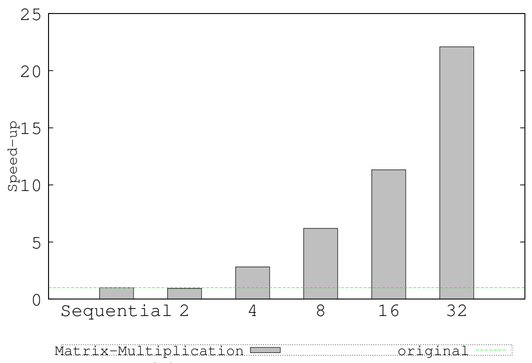
In this section we present some experiments to check the scalability and performance of `EMINENT`. We have used Milu [10], a well known mutation testing tool for the generation of mutants.

The experiments have been performed in a cluster that consists of 8 nodes interconnected through a Gigabit Ethernet network. Each node contains a Quad-Core Intel(R) Core(R) i5-3470

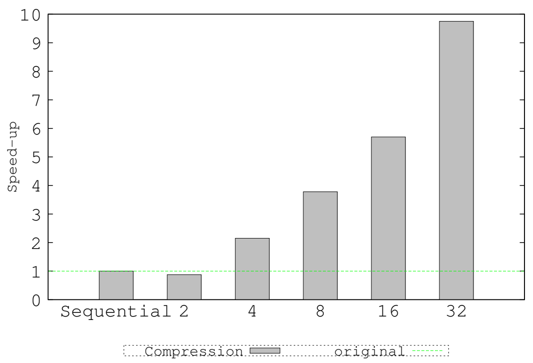
CPU at 3.4 Ghz with hyper-threading, 8 GB of RAM and 500GB HDD. In order to measure the scalability of the proposed algorithm, we have performed several executions with different number of processors.



(a) Image filtering



(b) Matrix multiplication



(c) Text compression

Figure 3: Speed-up of the applications using `EMINENT`

We have tested three different applications: *image filtering*, *massive computational application* and *text compression*. The first one applies filters to images. It uses 3 different filters to process images: *grayscale*, *median* and *saturation*. Initially, all the images are located in a database node with a total size of 2,5 GB. The master reads the image dataset from a remote database and distributes them among the worker processes using the communication network. Then, these images are filtered by the workers and saved in the local storage to check the I/O scalability. In this case, a test is given as a tuple $T =< I, F, O, C_{md5} >$ where I is the image to be processed, F is the filter to be applied and O is the filtered image. Finally, C_{md5} is a function that calculates the md5 hash of O. This value is used to compare the results obtained from the application of the test to the original program and each one of the mutants. This application and mutants were executed against 3200 test cases. The number of mutants generated to measure the test suite effectiveness was 250.

The second application performs a large number of operations in order to multiply matrixes, which means a huge computational load. The experiments performed over this application intended to analyse the computational scalability of the proposed algorithm. In this case a test is given as a tuple $T =< R, N, O, C_{md5} >$ where R is the seed used to build a pseudo-random matrix, N is the size of the matrixes, O is the result of the matrix multiplication and C_{md5}, as in the previous case, is a function that calculates the md5 hash of O. The test suite used to test this application contained 2000 tests and the number of mutants generated was 100.

The last application performs file compression using the *LZ4* algorithm. Initially, all the text files are stored in a remote repository composed of 1500 elements with a total size of 3 GB. Then, these files are compressed by the workers. The test cases are given as tuples $T =< F, O, C_{md5} >$ where F is the file to be processed, O is the result of the compression and C_{md5} is a function which calculates the md5 hash of O. In this case the test suite contained 1000 tests and a total of 200 mutants were generated.

Figure 3 shows the overall speed-up obtained by using `EMINENT` with these applications. The performance improvement is measured on the basis of a sequential execution. The applications were executed using 2, 4, 8, 16 and 32 processes. The greater the number of processes the greater the speed-up is. In all the cases, one of the processes acts as the master, which distributes the workload among the rest of the processes, which play the role of workers. Thus, the master process is not involved in the execution of the test cases. It is worth noting that the reduction observed in the performance during the experiments carried out with 2 processes with respect to the obtained during sequential execution, is due to the time spent in the communication between the master and worker processes.

Figure 3(a) shows the overall speed-up obtained during the testing of the *image filtering* application. The maximum achieved speed-up is close to 8. This performance reduction is because of the high volume of network and I/O traffic generated by this application, which acts as a system bottleneck in the database node. Since the master process does not execute tests, there is a significant increasing of performance when 2 and 4 processes are used, which means that 1 and 3 worker processes are executed, respectively. In this case, using 4 processes obtains a performance 2,85 greater than using 2 processes. By the contrary, the experiments using 4 to 32 processes show that the obtained performance slowly increases when the number of worker processes increases. This is due to the increase in the network traffic generated by obtaining the images from the database, which is shared by all the worker processes.

Figure 3(b) shows the results obtained in the experiments performed on the *massive computational* application. In this case the maximum achieved speed-up is 22. This fact is due to this application executes most of the operations in the CPU and sends small amounts of data between processes, which reduces the communication bottleneck. Consequently, the obtained performance when the number of processes varies from 4 to 32 increases almost linearly. Figure 3(c) shows the results corresponding to the *text compression* application. The maximum achieved speed-up is close to 10. Similarly to the image filtering application, the performance seems to be limited by the vast quantity of network traffic generated to obtain the text files from the remote repository.

With the aim of measuring the effectiveness of `EMINENT` we have compared it with a previous proposal to improve the performance of the mutation testing scheme, PEDRO algorithm. This approach presents the best results if we compare it with other works that have dealt with this issue. In order to carry out the comparison between these two algorithms, two types of execution grain were used. The PEDRO algorithm is based on mutant-level grain and `EMINENT` is based on test-level grain. We use the same applications but in this case the test suite generated for each of them contained 5000 test cases. The size of the data repositories used in the image filtering and text compression applications was increased up to 4 GB and 6 GB respectively. Figure 4 shows a comparative chart of the results obtained from both algorithms. All the experiments were performed with 32 processes in order to to maximise the parallelism level. In addition,

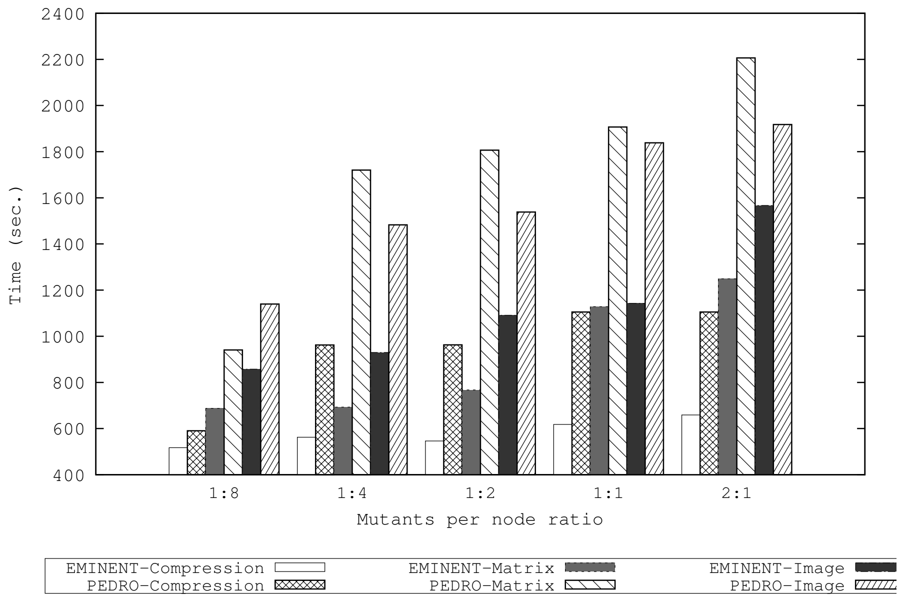


Figure 4: Results of comparison between EMINENT vs PEDRO

the executions were performed with different mutants per node ratio to check the scalability of both algorithms under different workload conditions. This ratio ranges from 1 mutant for each 8 nodes, to 2 mutants for each node. The chart shows that the test-level grain of EMINENT is more adaptable and suitable to maximise the exploitation of the computational resources than the mutant-level grain of PEDRO. In all the experiments carried out with EMINENT the performance obtained was better than the one showed by PEDRO. Overall, EMINENT scales better than PEDRO due to the execution time seems to grow slower in the experiments performed with EMINENT when the mutants per node ratio increases.

5 Conclusions and future work

We have presented a distributed algorithm, called EMINENT, designed to face the computational challenges associated with mutation testing. The main goal of this algorithm is to provide a scalable, dynamic and HPC-based method for reducing the execution cost by maximizing parallelism in this testing technique. The evaluation results show that EMINENT provides a better speed-up than the existing approaches in three different applications. These experiments have shown that EMINENT has a high adaptability level in heterogeneous computational environments where different usage levels of CPU, I/O and network are considered. Moreover, we have shown that the overall performance of the system systematically increases with the number of processes. This fact is a clear indicator of the scalability of our proposal.

As future work we plan to extend our proposal in order to parallelise the testing process of the original program so that we can reduce even more the time costs. Moreover, we will integrate other HPC techniques, such as an online compression layer, to reduce the loss of performance introduced by the network latency. Finally, we will adapt to our framework existing formal approaches to test distributed and timed systems [1, 7].

Acknowledgements

Research partially supported by the Spanish MEC project DArDOS (TIN2015-65845-C3-1-R) and the Comunidad de Madrid project SICOMORo-CM (S2013/ICE-3006).

References

[1] C. Andrés, M.G. Merayo, and M. Núñez. Formal passive testing of timed systems: Theory and tools. *Software Testing, Verification and Reliability*, 22(6):365–405, 2012.

[2] P.C. Cañizares, A. Núñez, M. Núñez, and J.J. Pardo. A methodology for designing energy-aware systems for computational science. In *Proceedings of the International Conference on Computational Science*, volume 51, pages 2804–2808. Elsevier, 2015.

[3] B. Choi and A.P. Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135–152, 1993.

[4] B. Fjukstad, J.M Bjørndalen, and O. Anshus. Embarrassingly distributed computing for symbiotic weather forecasts. In *Proceedings of the International Conference on Computational Science*, volume 18, pages 1217–1225. Elsevier, 2013.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[6] W. Grosso. *Java RMI.* O'Reilly & Associates, Inc., 1st edition, 2001.

[7] R.M. Hierons, M.G. Merayo, and M. Núñez. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, 25(1):35–62, 2012.

[8] S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A method for scheduling parallel loops. *Journal of Communications of ACM*, 35(8):90–101, 1992.

[9] L. Ismail and L. Khan. Implementation and performance evaluation of a scheduling algorithm for divisible load parallel applications in a cloud computing environment. *Software: Practice and Experience*, 45(6):765–781, 2015.

[10] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Proceedings of Practice and Research Techniques*, pages 94–98. IEEE, 2008.

[11] E.W. Krauser, A.P. Mathur, and Vernon J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[12] Y. Ma, A.J. Offutt, and Y. Kwon. Mujava: An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[13] A.P. Mathur and E.W. Krauser. Modeling mutation and a vector processor. In *Proceedings of the International Conference on Software Engineering*, pages 154–161, 1988.

[14] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top500 supercomputer sites, 2016. http://www.top500.org.

[15] A. Núñez, R. Filgueira, and M.G. Merayo. Sancomsim: A scalable, adaptive and non-intrusive framework to optimize performance in computational science applications. In *Proceedings of the International Conference on Computational Science*, pages 230–239. Elsevier, 2013.

[16] A.J. Offutt, R.P. Pargas, S.V. Fichter, and P.K. Khambekar. Mutation testing of software using a mimd computer. In *Proceedings of the International Conference on Parallel Processing*, pages 255–266, 1992.

[17] D.R. Penas, P. González, J.A. Egea, J. R. Banga, and R. Doallo. Parallel metaheuristics in computational biology: An asynchronous cooperative enhanced scatter search method. In *Proceedings of the International Conference on Computational Science*, pages 630–639, 2015.

[18] K. Qureshi and H. Rashid. A performance evaluation of rpc, java rmi, mpi and pvm. *Malaysian Journal of Computer Science*, 18(2):38–44, 2005.

[19] P. Reales and M. Polo. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *Proceedings of the International Conference on Software Maintenance*, pages 646–649. IEEE, 2012.

[20] P. Reales and M. Polo. Parallel mutation testing. *Journal of Software Testing, Verification and Reliability*, 23(4):315–350, 2013.

[21] V. Rego and A.P. Mathur. Concurrency enhancement through program unification: a performance analysis. *Journal of Parallel and Distributed Computing*, 8(3):201–217, 1990.

[22] I. Saleh and K. Nagi. Hadoopmutator: A cloud-based mutation testing framework. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 172–187. Springer, 2014.

[23] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *Proceedings of the International Conference of Software Engineering Conference and the ACM SIGSOFT Symposium*, pages 297–298. ACM, 2009.

[24] Statista. Media usage in an internet minute as of august 2015, 2015. Available at `http://www.statista.com/statistics/195140/new-user-generated-content-uploaded-by-users-per-minute/`.

[25] M. Towara, M. Schanen, and U. Naumann. Mpi-parallel discrete adjoint openfoam. In *Proceedings of the International Conference on Computational Science*, volume 51, pages 19–28. Elsevier, 2015.

[26] R.H. Untch, A.J. Offutt, and M.J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 139–148, 1993.